PARALLELISM IN DISTRIBUTED PROGRAMS: MEASUREMENT AND PREDICTION

by

Barton P. Miller

Computer Sciences Technical Report #574

January 1985

Parallelism in Distributed Programs: Measurement and Prediction

Barton P. Miller

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Rd.
Madison, WI 53705

Abstract

We have constructed a system for monitoring the behavior of distributed programs. The measurement system provides traces of the execution of programs and has been implemented on both the DEMOS/MP and Berkeley UNIX 4.2BSD operating systems.

Among the analysis techniques that have been applied to the measurement traces is the computation of the amount of parallelism, or concurrent activity, in a program. There are a number of interesting features of our parallelism measurements. First, the programs being measured can be run on a system that is concurrently running other programs. For most cases, system loading does not affect the measurements. Second, the measurement data can be used to predict the performance of the program in other configurations (e.g., different assignments of processes to processors). We can also predict the performance of the program in an ideal distributed environment (infinitely fast communications and unlimited processor resources). This prediction can be used as a upper-bound reference on the program's performance.

Research supported by the National Science Foundation grant MCS-8010686, the State of California MICRO program, and the Defense Advance Research Projects Agency (DoD) Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

1. Introduction

A common goal for creating a distributed program is to provide parallel execution. If a program can be subdivided into parts whose execution can proceed in parallel then the elapsed time for the program to complete can be decreased. A question of interest to the designer of such a program is how much parallel execution will occur in the program. There are several approaches to determining the amount of parallelism, ranging from analysis of the algorithm used in the program, to simulation of the program, to measurement of the program's execution.

We have constructed a system for monitoring the behavior of distributed programs. This system allows us to trace the behavior of a distributed program, providing data to analyze the program's behavior. We use this system to measure the amount of parallelism that occurs in the execution of a distributed program. We can determine the amount of parallelism in a program, even when our program executes on a computer that is currently executing other programs. In addition, our analysis of parallelism allows us to predict the amount of parallelism for the program running with a different assignment of processes to machines, varying machine loads, and different network characteristics. We can also determine an upper bound on the amount of parallelism possible for a given execution of a program.

The measurement and analyses presented in this paper apply to the performance of executing programs, as opposed to the performance of algorithms or simulation of the performance of programs. The path from algorithm, to simulation, to program, describes the development of the program. A complete development path would include the following steps. First, we develop a parallel algorithm and analyze its performance. Next, we construct an analytic model of the program, providing more details on the performance of the algorithm and an initial view of the structure of the program. Third, we build a simulation of the algorithm to measure its performance to provide more details of what the final program will do. Last, we measure the performance of the running program. The last step is necessary because the transition from algorithm (or simulation) to program often contains many surprises. We should determine the performance of the final product, if only to

verify the models and simulations, and that the program is behaving as desired. Also, there will be programmers (and this may be the more common case) that write programs that have not been simulated and had their algorithms simulated.

A common method of measuring parallelism or speed-up is to determine the performance of a program (or a simulation or an algorithm) on a single machine and compare that to the performance on multiple machines. This performance measure has been used in studying parallel algorithms [Baudet 78], distributed simulations [Chandy & Misra 78, Livny 85], simulations of interconnection networks [Fujimoto 83, Reed 1984], and multiprocessor architectures [Kuck 78].

In Section 2, we describe our model of computation and the measurement facility used to collect the data for the parallelism analysis. The data structures used for the analysis are described in Section 3. Section 4 describes the computation of parallelism, and various extensions. Section 5 outlines two studies that have been performed using our measurement facility and the parallelism analysis. The last section provides summary comments.

2. The Model of Computation and the Measurement Facility

This section provides a brief description of our model of distributed computation and the measurement system that was used to gather the data necessary for the parallelism analysis. A complete description of the measurement system can be found in [Miller *et al* 84, Miller 84, Miller 85].

2.1. Model of Computation

Our model of distributed computation (see Figure 2.1) is intended to be general enough to include a wide variety of systems. The model states that a process can only compute and communicate, and that communication is via messages. The only blocking activity in this model is a message receive. Processes do not share memory. The semantics of message passing may be synchronous or asynchronous, provide blocking or non-blocking operations, be buffered or unbuffered, and provide unidirectional or bidirectional message paths. The messages paths may be one-to-one or one-to-many; though the current parallelism analysis only supports one-to-one message paths. Most forms

of message passing (or interprocess communication) in contemporary operating systems fit this model.

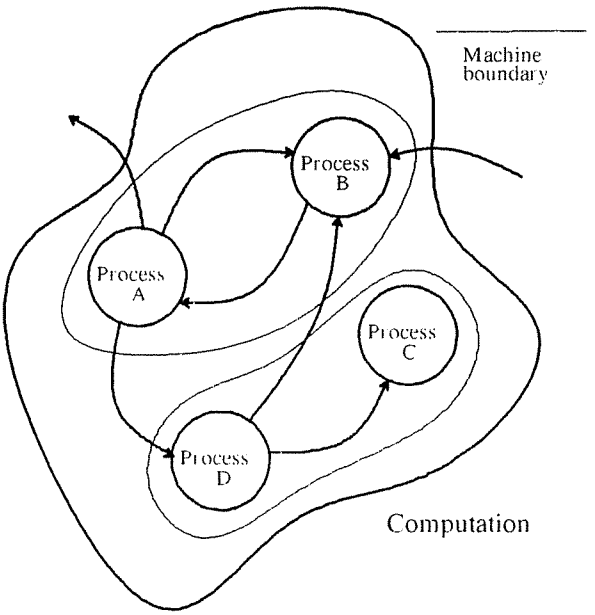

Figure 2.1: A Distributed Computation

2.2. Measurement Facility

Our measurement model is concerned with the *external events* in the life of a process. We define an *event* to be an action performed by a process in a computation. External events are those events in a process that affect other processes. Examples of external events are sending a message, creating a process, or destroying a message path. *Internal events* are those events not visible outside of a process, such as a process executing the statement

$$x := x + 1.$$

Our measurement system (called the Distributed Programs Monitor, or DPM) records the history of selected external events in the life of the processes in a computation. The data recorded about the occurrence of an event is called a *trace record* of the event (or more simply, a trace). In our measurement model, the detection of external events is referred to as *metering*. A trace is produced for each event that is detected. After the trace is produced, a decision is made whether or not to keep the trace. If the trace is kept, it can then be processed to provide results that are used to understand

the behavior of the process (and the overall computation). The measure of parallelism presented in this paper is an example of information that can be derived from the trace records.

A trace record is produced for each event occurring during the execution of a process. The trace records consist of two parts. The first part is a header which contains information common to all traces (including the event type). The second part of the trace is the body which contains information particular to the event. A trace record is show in Figure 2.2. The header of the event trace contains the fields in the following list. There are other fields in the header that are not relevant to the analysis of parallelism.

MACHINEID	The machine from which the trace came.
PROCTIME	The amount of CPU time used by this process up to the time this trace was generated. PROCTIME is independent of the load on the host system.
DATETIME	This is wall clock time. It cannot be assumed that the clocks on various machines are synchronized, so events happening at the same time on different machines may not have the same DATETIME field.
TRACETYPE	The type of event described by this trace.
LOADAVERAGE	This is the current load on this machine. Load is typically the number of ready processes, computed as an average over a recent time interval.

The trace body contains the fields particular to the trace event type. Figure 2.2 shows an event trace for a message send. This body identifies the sending and receiving process, and other fields particular to the operating system from which this example was taken (DEMOS/MP).

Header					*Body*				
MACHINE ID	PROC TIME	DATE TIME	TRACE TYPE	LOAD AVERAGE	FROM	TO	MSG LENGTH	CHANNEL	...
23	120	1234	Send	2.3	A	B	258	1	

Figure 2.2: Event Trace: Send Message

2.3. Implementation Notes

DPM is currently running on the DEMOS/MP [Baskett *et al* 77, Powell 77, Powell & Miller 83] and 4.2BSD UNIX [Joy *et al* 83] operating systems. The structure of these operating systems are quite different, yet the implementation of DPM on the two systems is similar. The DEMOS/MP version has been operational since September, 1983, and the 4.2BSD version since May, 1984. Examples given in this paper come from both of these implementations.

3. Computation Graphs

The DPM measurement system provides a stream of traces from the execution of a distributed computation. In particular, we use the trace of the message send and receive events. We must convert this trace stream to a more convenient data structure before we can proceed with the analysis of parallelism. The data structures described in this section are also used in other types of program analyses. Descriptions of these analyses can be found in [Miller 84] and [Miller 85].

3.1. Constructing the Graph

A process execution is a sequence of events. We represent a process as a list of external events, with the events as nodes, and with directed arcs joining the nodes showing the flow of execution. The arcs joining the events in a process are called the *process arcs*. Figure 3.1 shows a computation consisting of three processes with its events and process arcs. This lists of events is called a *computation list*.

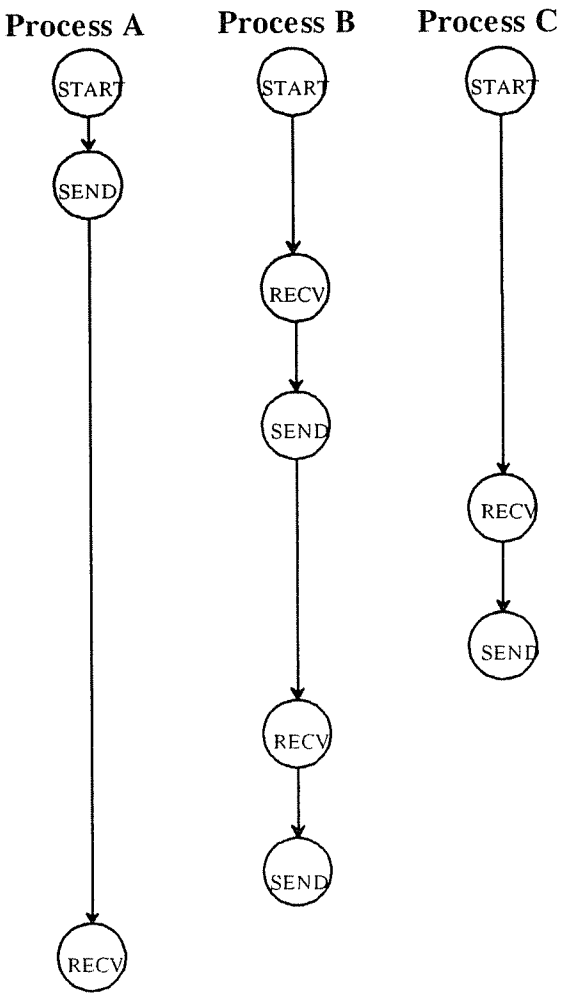

Figure 3.1: A Computation and its Events

There are interactions between processes in a computation. These interactions take the form of messages between the processes. The message interactions are added to the computation list to form a directed graph. This graph is called the *computation graph*. Each message is represented as an arc from the send event to the corresponding receive event (providing both events are present in the graph). An arc representing a message interaction is called a *message arc*. Figure 3.2 shows the sample computation with the message arcs added. Each node in the graph has an in-degree of at most two, and an out-degree of at most two. Each message has exactly one sender and one receiver. The computation graph is directed and acyclic. This is because the computation graph does not describe the program itself, but the history of one execution of the program; following the directed arcs through the graph reflects the passage of time.

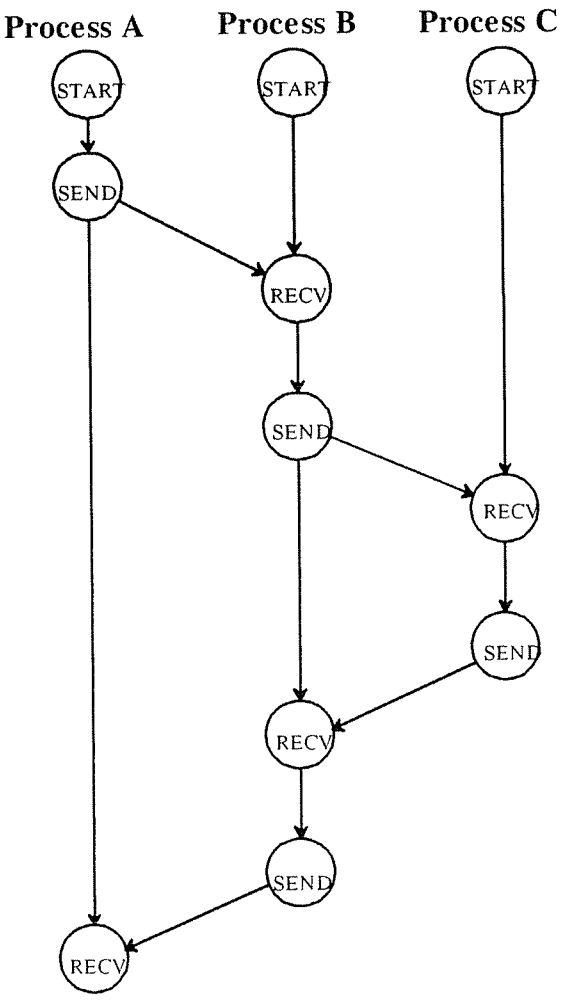

Figure 3.2: Computation with Message Arcs

3.2. Labeling the Graph

Given a computation graph, we label the arcs of the graph with the values that will be used in the analysis of parallelism. We first label the process arcs. The value on the label represents the amount of CPU time used by the process between the events. We next label the message arcs. The value on the message arcs represents the time required to delivery the message.

Each process, i, consists of a list of events, $e_{i,1}$, $e_{i,2}$, ... $e_{i,n}$. The trace record for event $e_{i,j}$ contains a PROCTIME$_{i,j}$ field, which is the total amount of CPU time used by process i up until event $e_{i,j}$. We label the arc between event $e_{i,j}$ and $e_{i,j+1}$ with $\Delta l_{i,j}$, where

$$\Delta t_{i,j} = PROCTIME_{i,j+1} - PROCTIME_{i,j}.$$

We next label the message arcs in the computation graph with the message delivery time. Message delivery time is a function, $MTIME(len, src, dst)$, where *len* is the length of the message in bytes, *src* is the machine originating the message, and *dst* is the machine receiving the message. Estimates of the time for a message delivery for the *MTIME* function can be obtained by sample measurements on the system where the traces were obtained, as described below. The message delivery time varies depending on which machines are involved in the message exchange, and length of the message. For each message arc in the graph, we compute the *MTIME* from these values and label the message arc with the result.

It is not possible to obtain the message delivery time from the message traces themselves, even though the traces have a field containing wall clock time. If a message was sent from a process residing on one machine to a process on another machine, the send and receive times would come from independent clocks. These clocks cannot be synchronized closely enough so that the times on the two machines can be compared. We obtained the values for the *MTIME* functions by measuring the message delivery times on the system that we are using. Measurements are made between the various types of machines and for different message lengths. These measurements need only be done once for each system. Message delivery time is calculated from a message sent from one machine to another, then back. The delivery time is considered to be a half of the round trip time.

These message delivery times are considered typical, and it is expected that there will be some variance in actual delivery times. If more accuracy is needed in calculating these values, the MTIME function can be extended to simulate more complex network behavior.

3.3. External Actions

Computations are not always self-sufficient. Processes in a computation may need to interact with other processes or parts of the host system that are not part of the computation being studied. There is no way for us to analyze activities that happen outside the available trace data. The problem becomes important when some process in the computation must wait for a response from an external

process (or part of the host system). Figure 3.3A shows that the time through the external process could be longer than the process time from the SEND to the RECV. If the communication with the external process is a synchronous request, the process time between the SEND and RECV would be close to zero, and the time taken in the outside process becomes important. We can model this problem by modifying the computation graph as in Figure 3.3B. The label of the new arc would be the difference in elapsed time (not process time) between the send and receive event.

System loads can affect the elapsed time between the SEND and RECV events. This is true for the case where the external process is sharing a machine with our computation, and for the case when it is on a separate machine. Therefore, the elapsed time can only be considered an approximation to the path length through the external process.

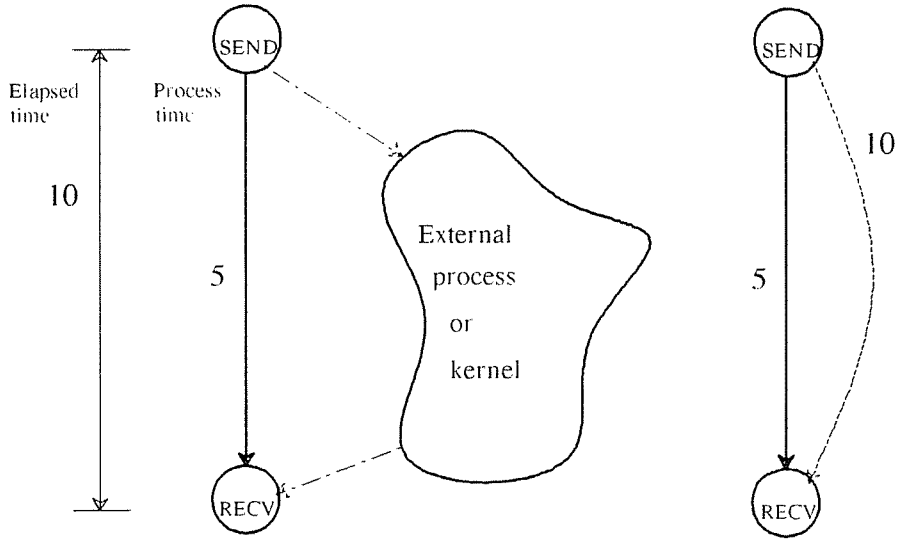

Figure 3.3: Computation Graph with External Event

4. The Parallelism Factor

Given the labeled computation graphs, we are ready to use these graphs to compute the amount of parallelism in the execution of the program represented by the graphs. We first define our metric

of parallelism, **P**. This definition gives the basic method of computing the amount of parallelism. We next consider the problems of calculating the amount of parallelism when processes share machines, calculating the effect of other loads on the system, and projecting the best case (upper bound) performance for the program's execution.

4.1. Calculating P

To measure parallelism in a computation, we first need to define parallel execution. This definition will be in terms of the trace events that we measure. Each executing process consumes machine cycles. We define *process time* for process i, t_{proc_i}, to be the number of units of CPU time that a process consumes over its lifetime, or

$$t_{proc_i} = \sum_j \Delta t_{i,j}.$$

The *total time*, T, for a computation is the sum of the process times for all the processes in the computation.

$$T = \sum_i t_{proc_i}.$$

In a graph for a computation with N processes, there are exactly N nodes with an in-degree of zero. These nodes represent the start of computation for each process and are called *initial nodes*. The graph also contains at most N nodes with out-degree of zero. These nodes represent the termination of computation and are called *termination nodes*. We define the *maximum path* to be the directed path from an initial node to a termination node that gives the greatest weighted path length. The value of this greatest weighted path is $t_{\max}$.

For a given execution of a computation, if T equals $t_{\max}$, then there has been no parallel execution. This would be the case if the entire computation were run on a single machine; the machine could execute the computation in no less time than the total process time used (T). If t_{proc_i} equals $t_{\max}$, for all i, then we are achieving 100% parallel execution; the computation has been split into uniform size pieces (processes) and all the pieces are executing concurrently.

Thus, we define the parallelism factor, **P**, to be

$$\mathbf{P} = \frac{T}{t_{\max}}.$$

The case where there is no parallel execution gives a **P** of 1, and the case of 100% parallel execution gives a **P** of N, for a computation consisting of N processes. If all the processes in a computation were executing on a single machine, and there were no interactions between the processes, we would have a **P** of 1. It is likely that if the processes were in the same computation, there would be some interaction. Given interaction between the processes, execution on a single machine, and non-zero message delivery time, it is possible to have a **P** of less than 1.

4.2. CPU Sharing and Contention

The calculation for **P** in the preceding section assumes that each process in a computation resides on its own machine, but this is not always the case. When more than one process executes on a machine, there is competition for use of the CPU. The number of processes competing for the CPU varies over the execution of the computation. Several factors determine how many processes are running. The first factor is how many processes are currently assigned to the machine. Other factors are process creation, termination, and processes migrating from their current location. The number of processes currently assigned to a machine forms the upper bound for the number that are able to run at any given instant.

Given the number of processes that could be run (the number currently on a particular machine), the problem is to determine when a process is running and when it is blocked from execution. A process is blocked from execution when it is attempting to receive a message that has not yet arrived. Figure 4.1 shows the sample trace of a computation, and we will consider the case in which all three processes execute on the same machine. Before the time at which the indicated SEND operation for process A occurs, process B cannot continue past the indicated RECV. If process B reaches this event before the message is sent (and arrives), then process B is blocked, thus reducing by one the number of processes that are contending for the CPU.

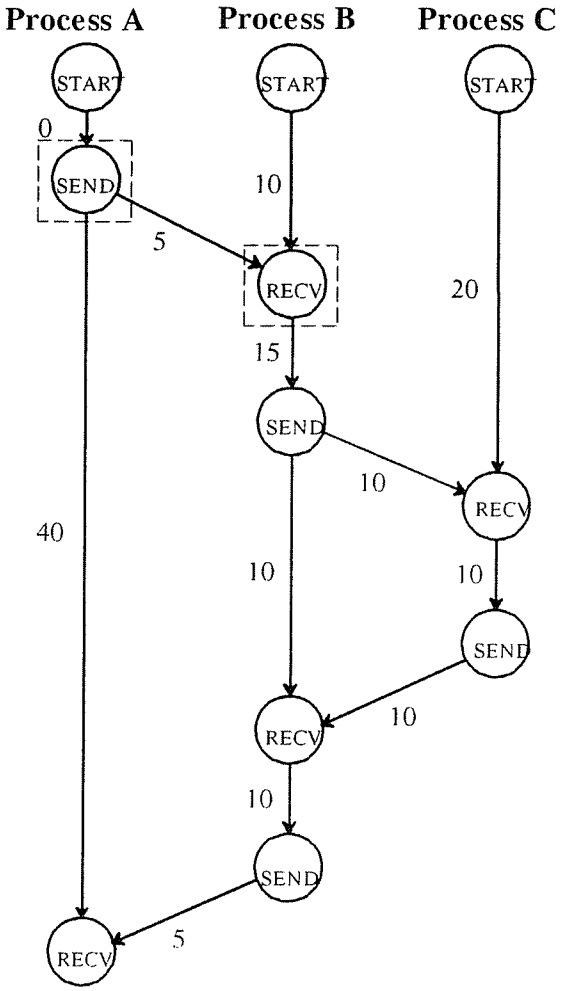

Figure 4.1: Computation with Corresponding SEND/RECV Events Marked

The algorithm that factors in the CPU sharing relabels the process time arcs with a new weight calculated from the number of processes currently able to run. Figure 4.2 shows the sample computation graph redrawn with concurrent time intervals (*slices*) for the three processes. Note that the time scale on the left side of the figure is elapsed time (for each process, if it were on its own machine).. For a given interval of t units of time, with k processes currently able to run, the time interval is relabeled with a value of $k \times t$. A process arc is labeled with the sum of the new values for the intervals during which the process was not blocked. There is a separate value of k for each machine, and these values changes over the life of the computation. If a process moves to a different machine during its execution, it contributes its load to the k for the new machine. Figure 4.3 shows the sample graph after the relabeling has been completed. The calculation of **P** from this relabeled

graph give a value that reflects the actual execution of the program. This part of the analysis assumes that the machines are processor sharing; i.e., using round-robin sharing of the CPU with a microscopic time slice. The complete algorithm for relabeling the graph is given in [Miller 84].

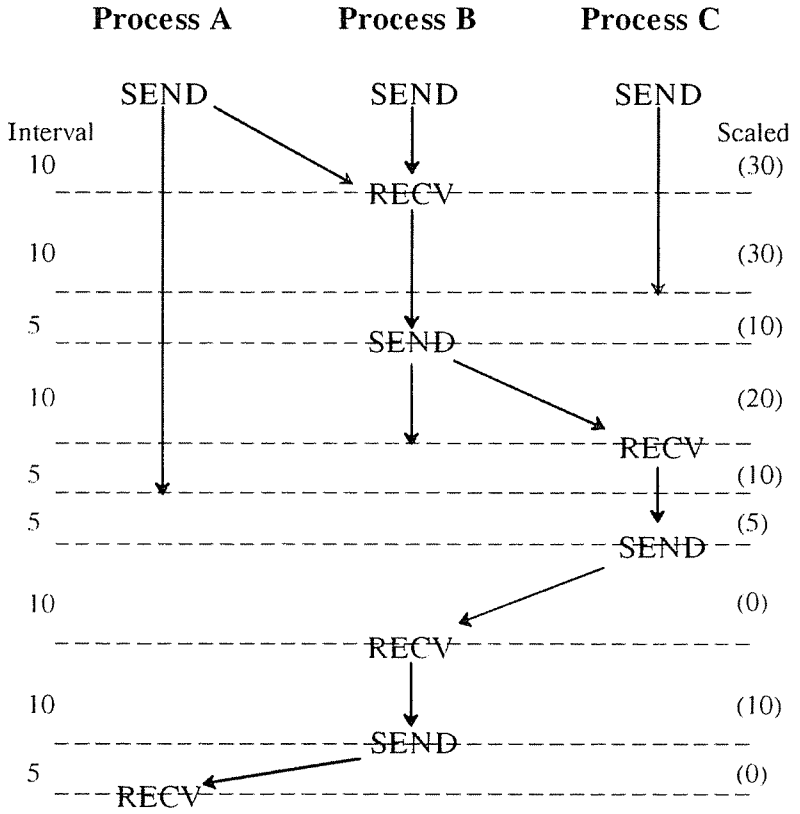

Figure 4.2: Computation Showing Intervals of CPU Sharing

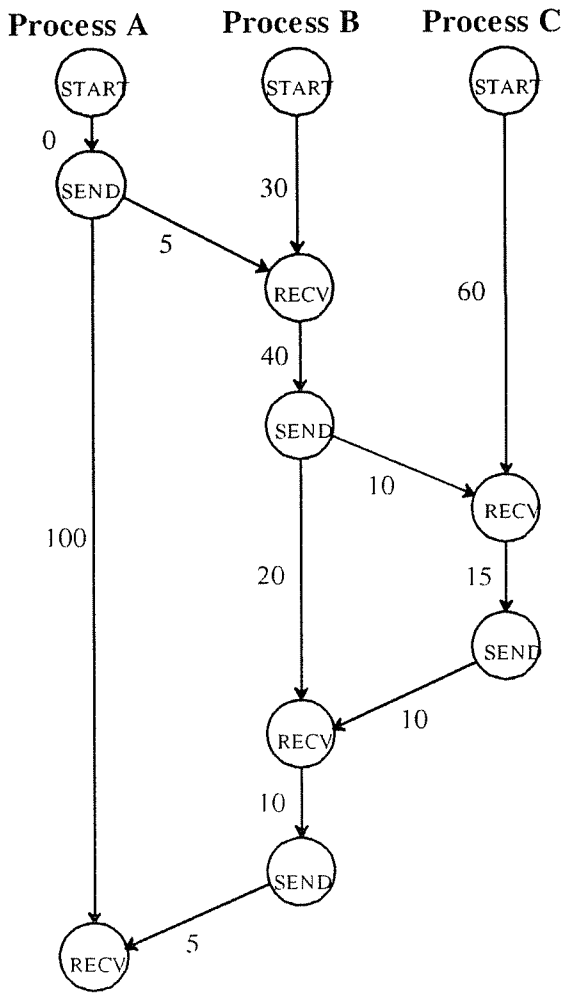

Figure 4.3: Computation with Relabeled Arcs

4.3. Projections and Upper Bounds

The parallelism analysis computes values for **P** based on an assignment of processes to machines and the labeling of the computation graph. The assignment of process to machines determines the *src* and *dst* parameters to the *MTIME* functions and which processes will compete for the same CPU. When we compute **P**, what assignment of process to machines should we use? The natural decisions to take the actual assignment from the execution of the program. This information is available in the trace data. However, we are not restricted to calculating **P** for the actual assignment. Processes can be assigned (in the analysis) to other configurations for the parallelism analysis. For each new assignment of processes to machines, new weights must be assigned to the

message arcs and the longest path calculations must be repeated.

The ability to vary the configuration allows the programmer to predict the performance of his/her program in different situations. If a program consists of processes that spend most of their time computing and very little time communicating, then it would probably be best to give each process its own machine. If two (or more) processes communicate frequently, then these processes may be best on the same or nearby machines. The programmer can examine many different configurations without having to re-run the program for each configuration. In addition, we can vary the values produced by the *MTIME* function to reflect different communications networks. This is similar to network simulators such as those of Fujimoto [Fujimoto 83] and Reed [Reed 1984].

There is another type of performance prediction that can be done with the parallelism analysis. First, we make *MTIME* a constant function whose value is always zero (i.e, instantaneous message delivery), and assign each process in a computation to its own machine. Then, we calculate the new value for t_{max} and $\mathbf{P}$. This calculation results in the maximum value for $\mathbf{P}$, called $\mathbf{P}_{max}$. Neither the addition of machines nor a faster communications media could result in a greater value. $\mathbf{P}_{max}$ gives a reference for the best possible performance of the execution of the program being measured. If better performance is needed, the program must be rewritten or a faster computer must be used.

There is the implicit assumption in the preceding discussion that the reassignment of processes to machines will not change the execution order of events. We must make this assumption because our results come from analyzing program executions, rather than the programs themselves. This assumption is valid if the order of interactions between the processes in the computation is deterministic. This is not always the case. A computation structured as a server, receiving requests from many independent sources, would not fit the deterministic assumption.

We can make a slightly weaker assumption about our computations. That is, that each process in the computation is deterministic with respect to its inputs. This means that a process will behave the same way independent of when it receives a given input. When processes are reassigned (in the parallelism analysis) to different machines than those on which they were really executed, the order

of execution of events in the computation could change. Given our weaker assumption about determinism in a process, this change of execution order should not have a large effect on the overall computation.

This weaker assumption does not always hold. The order of interaction can affect such things as the order of inserting elements into a database. However, the overall effect of these factors should be small enough in actual computations that the results for reassignment will be valid. Section 5 provides experimental results from the DEMOS/MP file system that support this argument.

There are also algorithms that behave much differently given a different machine assignment. An example of this is α-β search [Akl1980, Finkel & Fishburn 82]. This is a search problem that is partitioned, with each of several processes performing part of the search. The first process to succeed in the search notifies the others that they can halt. For these algorithms, the parallelism analysis can report the actual amount of parallelism, but it may not be accurate for performance prediction in other configurations.

4.4. System Loads

The parallelism analysis does not consider that the computation being measure may share a machine with processes that are not part of the computation. The analysis is independent of other loads on the machines because it is based on process time. However, there may be cases where we want to predict the behavior of our computation on machines that are being shared by other computations. This affect of other computations can be added to the parallelism analysis in several ways. The simplest way is to estimate what percentage of machine cycles for each machine is available to our computation. This percentage is then used as a scaling factor when we relabel the computation graphs.

A more accurate method uses the machine load information (from the LOADAVERAGE field) included in the trace message header of each event trace. As the load on each machine changes over time, this value can be used as scaling factor that varies over the life of the computation.

4.5. Utilization

While **P** appears to be a useful metric for evaluating the performance of a distributed computation, this metric alone is not sufficient. By using **P** as our only evaluation criterion, we have stated: faster is better. This view of performance dictates that an increase in parallelism is our only goal.

It may also be important to know how well we are using our computing resources. This can be stated as: how much of our available computing resources are we using, or how much of the machines (CPU's) involved in the computation are we utilizing? We can compute CPU utilization from the parallelism factor. We calculate utilization, ρ, as

$$\rho = \frac{\mathbf{P}}{M}$$

where M is number of machines used for in the computation.

5. Sample Results

The analysis of parallelism presented in this paper is based on the measurement of real programs. In this section we present the results of two studies using data collected by DPM and subjected to our parallelism analysis. The first study is of the development of a parallel program to find solutions to the Traveling Salesman Problem. The second study is an analysis of the parallelism in the DEMOS/MP file system.

5.1. Study 1: The Traveling Salesman Problem

This study is intended to evaluate how well a particular algorithm performs at varying levels of concurrency. The Traveling Salesman Problem (TSP) [Christofides 79] is a well studied problem and can be decomposed into parts that can be computed concurrently. A complete description of our study of the problem is given in [Lai & Miller 84], and we present only a summary of the results here.

Briefly described, TSP is the problem of finding the minimum cost for visiting each of N cities exactly once, given the costs of traveling from one city to another. If we consider the cities to be the N vertices, V, in a graph, and the paths between cities (with associated travel costs) to be the M

edges, E, then the problem is described by the directed graph $G = (V, E)$. The solution to TSP is the Hamiltonian circuit with the minimum cost.

We measured a TSP program running on 4.2BSD UNIX operating system. The computation was structured as a collection of processes executing concurrently and communicating via messages. The TSP implementation used is similar to the one described in [Mohan 82]. The minimum path is found by dividing the set of possible paths into smaller sets of paths. The path that appears to provide the best solution is picked from this smaller set of paths. If this choice proves not to be best, then the algorithm backtracks and tries another choice. This procedure is repeated until a solution is obtained. The cost for a single choice within a set of paths can be computed independently. The computation is performed by p processes, where $p \leq N$. This means that up to p simultaneous computations can be performed.

If there is no backtracking in the computation (i.e., if our choice of the best solution was always correct), then the computation is first divided into $N-1$ pieces and the $N-1$ pieces are computed concurrently. The computation is then divided in $N-2$ pieces and the procedure is repeated. The entire operation continues until a complete path is found.

Our computation is structured into a *controller* process and up to p *server* processes. The controller is responsible for dividing the computation into pieces to be executed by the servers. The main data structure used by the TSP computation is an $N \times N$ matrix. When the computation starts, a copy of this matrix is sent to each server. After this initial distribution, the only communications between the controller and the servers are messages from the controller instructing the servers to compute some part of the problem (a subset of the matrix), and the messages from the servers to the controller indicating completion of their assigned computation.

We implemented a preliminary version of the TSP program (called "Version 1"), and ran this program varying the size of the problems (number of vertices in the graph, or equivalently, the size of the matrix) and the number of processes used to compute the solution. The problem was computed for problem sizes from 8, 12, 16, and 20 vertices. For each size, we computed the solution

using 4, 8, 12, and 16 server processes.

Three values of **P** were calculated for each of the above variations: **P** with one server process on each machine, **P** with two server processes on each machine, and $\mathbf{P}_{max}$. These values are plotted in Figure 5.1, Figure 5.2, and Figure 5.3, respectively.

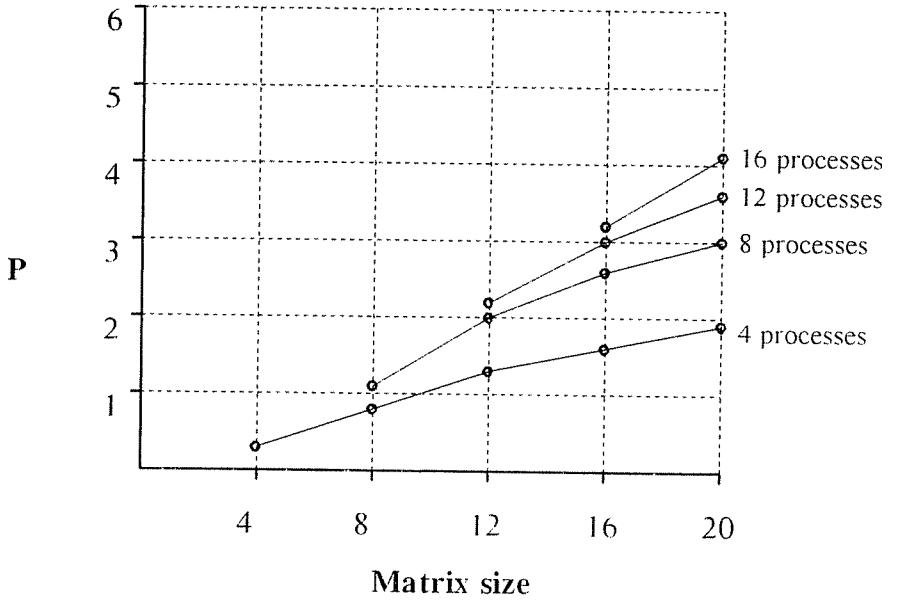

Figure 5.1: Parallelism vs. Matrix Size, One Process per Machine

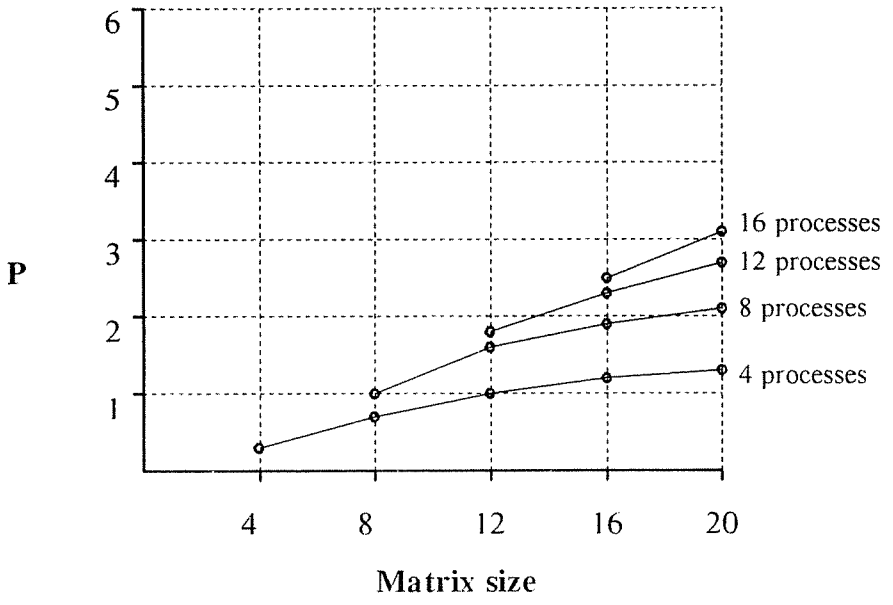

Figure 5.2: Parallelism vs. Matrix Size, Two Processes per Machine

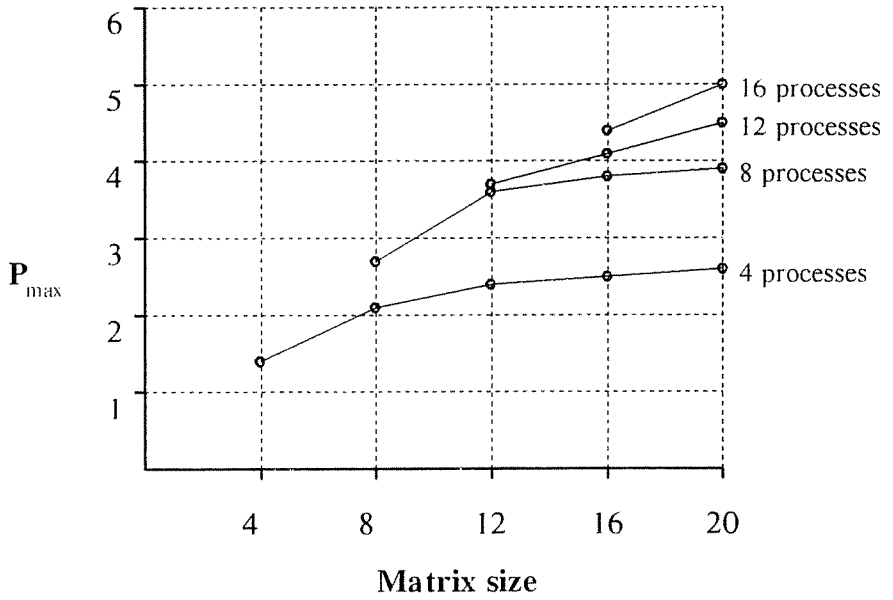

Figure 5.3: Upper Bound Parallelism of the Non-Blocking TSP Program

Version 1 of the TSP program was then modified based on the parallelism analysis and other performance metrics (such as CPU consumed by each process and message counts, also available from DPM). The second version of the TSP program cached intermediate calculations so that if backtracking was necessary certain repetitions of calculation could be avoided. The code in the mas-

ter process was also rewritten to be more efficient. This version (called "Low-Node Caching") was run and monitored in the same manner as Version 1. The Low-Node Caching version was further modified so that the master process would overlap requests to the slave processes by initially sending multiple requests to each slave. This avoided some of the time lost waiting for communication delays. This version (called "Overlapping Requests") was also run and monitored. The results of these measurements are summarized in Figure 5.4. The graph compares the values of **P** for the three versions of the TSP program. The values are plotted for one server process per machine, two server processes per machine, and $\mathbf{P}_{max}$. Each case is the solution to a 16 × 16 matrix. This graph shows the performance increase for each subsequent version of the program. It provides a concrete indication of the change (improvement) as we modified the program.

P was not enough to evaluate the three versions of the TSP program. In our development of the program, we also used T, the total CPU time. When comparing values of T for the three versions, the change from Version 1 to Low-Node Caching decreased T by 15-20% (for the problem sizes that were compared), and the change from Low-Node Caching to Overlapping Requests brought no significant change. The first change produced a version with not only more parallel activity, but one that used less CPU resources. The second change also produced more parallel activity, but used the same amount of CPU resources.

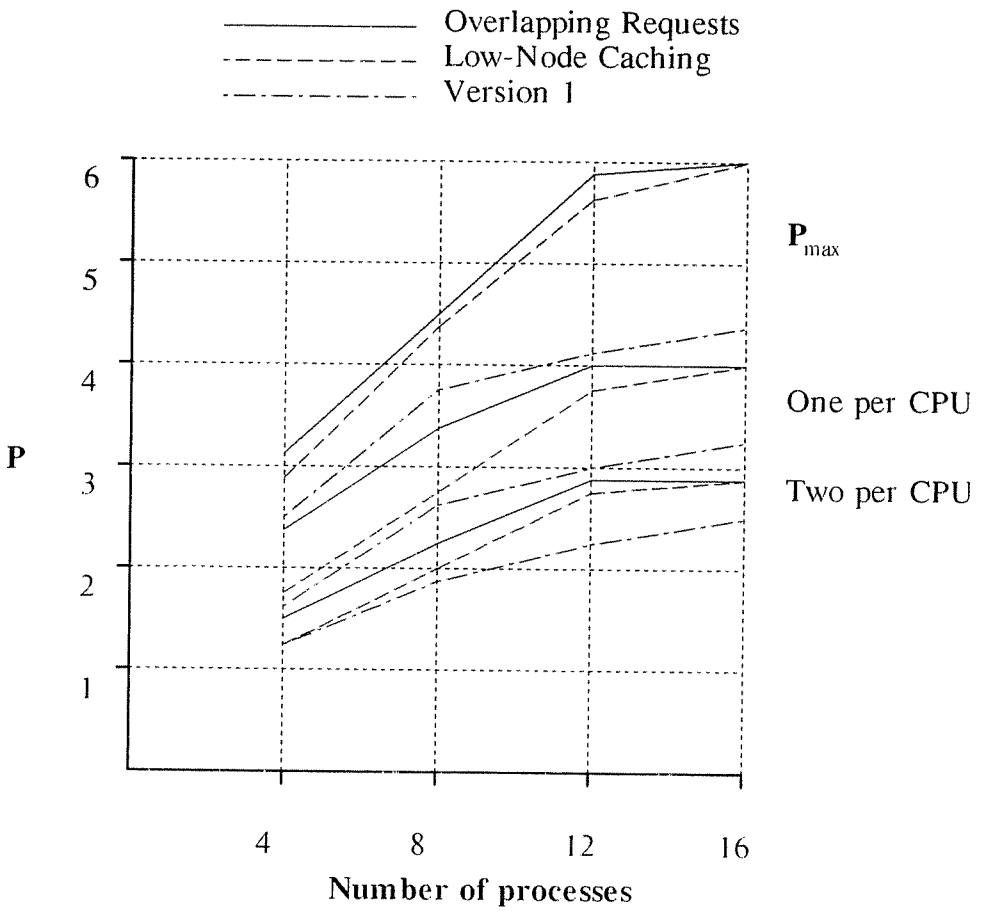

Figure 5.4: Comparison of the Three Implementations - 16 × 16 matrix

5.2. Study 2: The DEMOS/MP File System

The DEMOS file system [Powell 77] provides an interesting case study for several reasons. The first reason is that it is a multiprocess computation. It consists of over 20,000 lines of high-level language code organized into four processes. Secondly, it performs a complex function used by a significant portion of the system. The file system interacts with both user (non-system) processes and with the operating system kernel. The third reason is that it is long lived, since it is available to be studied for as long as the operating system is running. A fourth reason is that it provides a program of significantly different structure than the TSP study. A summary of the results of the DEMOS/MP file system study are presented here. A complete discussion is given in [Miller 84].

DEMOS/MP is a message-based operating system, and the file system is structured as a file server; processes send request messages to one of the file system processes and receive response

messages when the service is accomplished. The file system processes may reside on different machines. The processes are the Request Interpreter, Directory Manager, Buffer Manager, and Disk Interface.

Our study examined the amount of parallelism in the file system under increasing loads (number of user processes making requests to the file system). The measurement environment was the DEMOS/MP operating system running on seven machines. The Disk Interface and Buffer manager were on one machine, and the Request Interpreter and Directory Manager were on another machine. Measurements started at the beginning of the file system's execution. This means that the file system's initialization activity was included. The user processes that made requests to the file system were active on the various machines over the period of the test. All user processes were started at the same time and the test was complete when these processes terminated. The user processes were test programs of two different classes. Each of these classes performed a different mix of computation and file I/O.

The first class of user processes performed repeated sequential accesses to a file; first writing the entire file, and then reading it. The second class of user processes did repeated random accesses to a file. These accesses consist of both reading and writing. The user processes start at the beginning of the test and terminate when they have completed their task. There were no other users during this test.

The test was repeated for increasing numbers of user processes (from 1 to 7) simultaneously making request to the file system. In each case, the file system was monitored by DPM and **P** was calculated from the event traces. Figure 5.5 summarizes the results from these measurements. This graph shows the saturation point of the file system (4 user processes). The saturation point can be interpreted as the maximum number of requests on which the file system can work simultaneously.

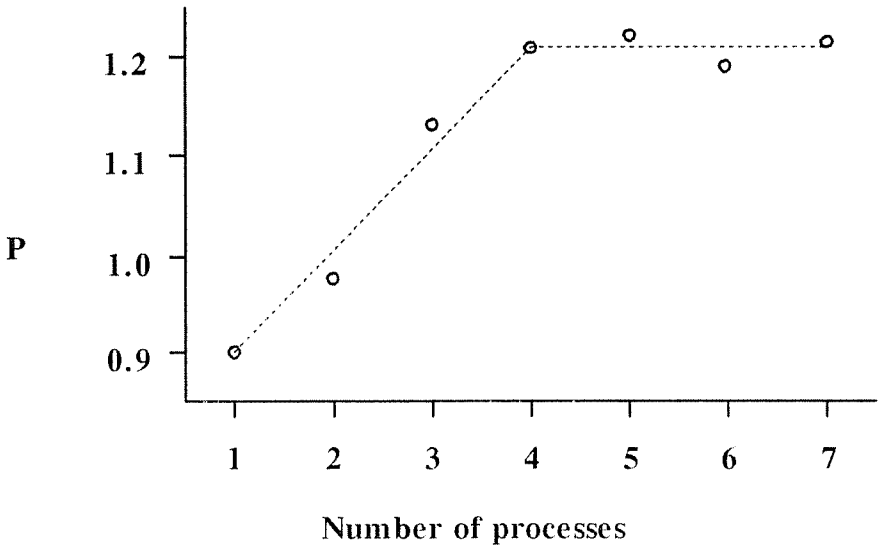

Figure 5.5: DEMOS/MP File System: P vs. # of User Processes

An additional measurement was performed with the DEMOS/MP file system. This was to measure the accuracy of the parallelism factor, **P**, as a predictor of performance. When we measure a computation, traces are collected from processes that execute on particular machines. Parallelism analysis gives us the ability to compute the amount of parallelism in a program for some assignment of processes to machines. Usually, we start by assigning (in the analysis) the processes to the machines on which they actually ran. This gives us the **P** for the *measured* configuration. Using the same trace data, we can assign (in the analysis) processes to different machines and recompute **P**. This gives us the *predicted* value for **P** if we had actually run the processes in this new configuration. The question is how well the predicted values of **P** match what would really occur.

Data was collected for the file system running in each of three configurations (shown in Figure 5.6ABC). The measurements were made and the parallelism factor was then calculated for each of the three configurations. The data for each configuration was then used for two more analyses. For example, the data collected from the configuration in Figure 5.6A was analyzed (and the value of **P** obtained) using the machine assignment from the configurations in Figure 5.6B and C to predicted the value for **P**. The same was done for the other two configurations. The result was nine values for **P** in a three by three matrix. Figure 5.7 shows the percent differences between the predicted values

of **P** and the values calculated for the measured configuration. For example, data was collected for configuration A, and **P** was calculated from this data based on the measured configuration. **P** was also calculated from data collected for configurations B and C, but assigning (in the analysis) the processes to machines as in configuration A. The results were differences of -3% and 2% between the measured and predicted values. Similar results were obtained for all of these analyses. These variations show no excessive loss of accuracy due to analyzing the trace data in configurations different from that in which it was collected. These results are not conclusive, but they provide an indication of the validity of the predictive part of our parallelism analysis techniques.

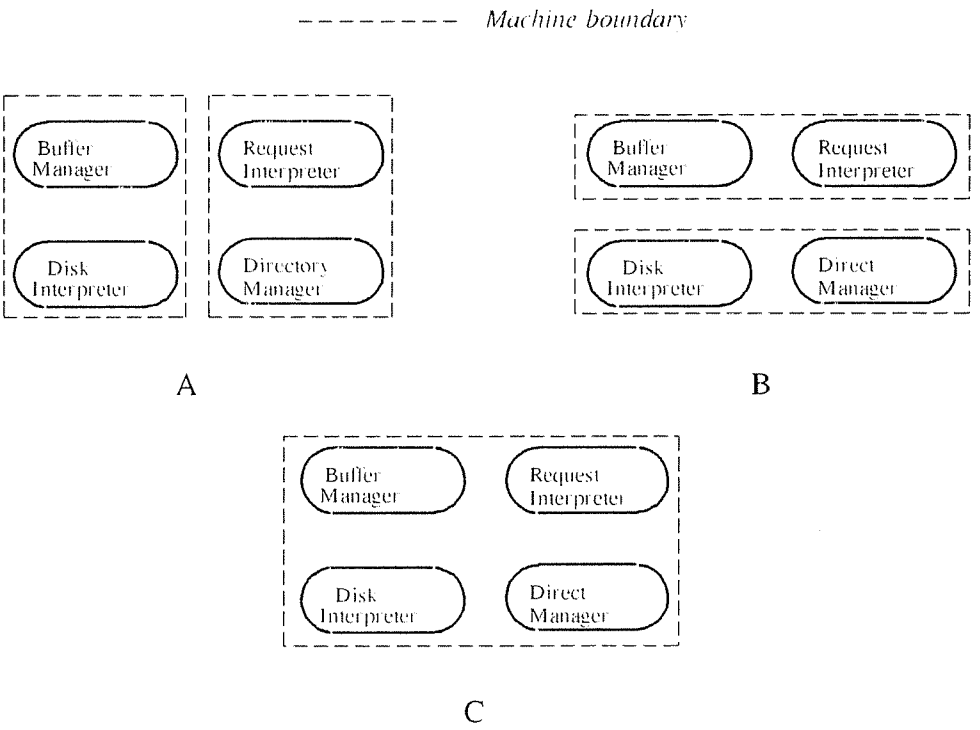

Figure 5.6: Three File System Configurations

		Predicted Configuration		
		A	**B**	**C**
Measured Configuration	**A**	-	-3%	2%
	B	4%	-	3%
	C	2%	-2%	-

Figure 5.7: Variation in P for the Three Configurations

6. Conclusion

Parallelism is an important metric when considering the performance of a distributed program. We have provided a means for programmers to determine the amount of parallelism in their programs, and to do so on machines shared with other users. Our parallelism analysis provides the ability to predict the program's performance for different assignments of processes to machines (even dynamically moving processes). The upper bound, $\mathbf{P}_{max}$, provides a reference for the limits of the performance of a program. This is important in order to know when to stop attempting minor changes (such as different assignments of processes to machines) and when to change the structure of the program or try a new algorithm.

Knowing **P** for a given program is not enough. Consider the case where we have two programs, *A* and *B*, that calculate the same function. We discover that when executing on 10 machines, program *A* has a **P** of 9.5 and that program *B* has a **P** of 1.5. It is tempting to say that program *A* is clearly better than *B*. But other performance metrics must be considered, such as machine utilizations or total CPU time used by a program. In our example, if we discovered that program *A* uses 1,000 CPU seconds and *B* uses 10 CPU seconds, we may decide to use *B*, even though it exhibits less parallel activity.

In general, it is important to be able to determine the performance of the programs that we write, and the parallelism metric is useful as one measure of performance for distributed programs.

References

[Akl1980]
S. G. Akl, D. T. Barnard, and R. J. Doran, "Simulation and Analysis in deriving Time and Storage Requirements for a Parallel Alpha-beta Algorithm," *Proc. of the 1980 Int'l Conf. on Parallel Processing*, pp. 231-234 (1980).

[Baskett *et al* 77]
F. Baskett, J. H. Howard, and J. T. Montague, "Task Communcations in DEMOS," *Proc. of the Sixth Symp. on Operating Sys. Principles*, Purdue, (November 1977).

[Baudet 78]
G. M. Baudet, "The Design and Analysis of Algorithms for Asynchronous Multiprocessors," Ph.D. Dissertation, Technical Report CMU-CS-78-116, Carnegie-Mellon University (April 1978).

[Chandy & Misra 78]
K. M. Chandy and J. Misra, "Specification, Synthesis, Verification, and Performance Analysis of Distributed Programs: A Case Study: Distributed Simulation," Technical Report TR-86, University of Texas, Austin (November 1978).

[Christofides 79]
N. Christofides, "The Traveling Salesman Problem," pp. 131-149 in *Combinatorial Optimization*, ed. Christofides, Miugozzi, Topf, and Saudi, Wiley (1979).

[Finkel & Fishburn 82]
R. A. Finkel and J. P. Fishburn, "Parallelism in Alpha-Beta Search," *Artificial Intelligence* **19** pp. 89-106 (1982).

[Fujimoto 83]
Richard M. Fujimoto, "SIMON: A Simulator of Multicomputer Networks," Technical Report UCB/CSD 83-140, University of California, Berkeley (September 1983).

[Joy *et al* 83]
W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher, "4.2BSD System Manual." Computer Systems Research Group Technical Report, University of California, Berkeley (July 1983).

[Kuck 78]
D. J. Kuck, *The Structure of Computers and Computations*, John Wiley & Sons, New York (1978).

[Lai & Miller 84]
N. Lai and B. P. Miller, "The Traveling Salesman Problem: The Development of a Distributed Computation," Technical Report UCB/CSD 84/212, University of California, Berkeley (October 1984).

[Livny 85]
M. Livny, "A Study of Parallelism in Distributed Simulation," *Proc. of Distributed Simulation 1985*, San Diego, Calif., (January 1985).

[Miller *et al* 84]
B. P. Miller, S. Sechrest, and C. Macrander, "A Distributed Program Monitor for Berkeley Unix," Technical Report UCB/CSD, University of California, Berkeley (September 1984).

[Miller 84]
B. P. Miller, "Performance Characterization of Distributed Programs," Ph.D. Dissertation, Technical Report UCB/CSD 85/197, University of California, Berkeley (May 1984).

[Miller 85]
B. P. Miller, "A Measurement System for Distributed Programs," in preparation (1985).

[Mohan 82]
J. Mohan, "A Study in Parallel Computation -- the Traveling Salesman Problem," Technical Report CMU-CS-82-136, Carnegie-Mellon University (August 1982).

[Powell 77]
M. L. Powell, "The DEMOS File System," *Proc. of the Sixth Symp. on Operating Sys. Principles*, pp. 33-42 Purdue University, (November 1977).

[Powell & Miller 83]
M. L. Powell and B. P. Miller, "Process Migration in DEMOS/MP," *Proc. of the Ninth Symp. on Operating Sys. Principles*, pp. 110-119 Bretton Woods, N.H., (October 1983).

[Reed 1984]
D. A. Reed, "The Performance of Multimicrocomputer Networks Supporting Dynamic Workloads," *IEEE Trans. on Computers* **C-33**(11) pp. 1045-1048 (November 1984).